Project 1: Threads

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
## Pintos Kernel (1)

- **The current Pintos kernel**
  - There is only one address space
  - There can be a number of threads running in the kernel mode
  - All the kernel threads share the same address space

<table>
<thead>
<tr>
<th># threads per addr space:</th>
<th># of addr spaces:</th>
<th>One</th>
<th>Many</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>One</td>
<td>MS/DOS Early Macintosh</td>
<td>Traditional UNIX</td>
</tr>
<tr>
<td>Many</td>
<td>The current Pintos</td>
<td>Mach, OS/2, Linux, Windows, Mac OS X, Solaris, HP-UX</td>
<td></td>
</tr>
</tbody>
</table>
**Address space**

- Up to 64MB of physical memory
- The kernel maps the physical memory at PHYS_BASE (0xc000 0000)

```c
static inline void *ptov (uintptr_t addr) {
    return (void *) (paddr + PHYS_BASE);
}
static inline uintptr_t vtop (void *addr) {
    return (uintptr_t) vaddr -
    (uintptr_t) PHYS_BASE;
}
```
Kernel thread

- The kernel maintains a TCB (Thread Control Block) for each thread (struct thread)
- Created using thread_create()

```c
#include <thread.h>

struct thread {
    // thread structure
} tcb;

// create a new thread
int thread_create(const char *name, int priority, thread_func *function, void *aux) {
    // allocate a page (4KB) for thread stack
    // initialize TCB
    // add TCB to the run queue
    // return the corresponding tid
    return tid;
}
```

- Allocate a page (4KB) for thread stack
- Initialize TCB
- Add TCB to the run queue
- Return the corresponding tid

- The function thread_current() returns the pointer to the TCB of the current thread
Pintos Kernel (4)

- TCB (Thread Control Block)

```
struct thread
  tid
  status
  priority
  ...
  allelem
  elem
  ...
  magic

struct thread *running_thread()
{
  Get %esp;
  return (%esp & 0xffffffff000);
}
```
**Thread states**

- Refer to Appendix A.2: Threads
Pintos Kernel (6)

- Ready queue

```
struct thread{
    ....
    ....
    struct list_elem elem
    ....
}
```

```
all_list
```

```
list_entry
through offset
```

```
ready_list
```

```
tid = 1
THREAD_RUNNING
allelem
elem
...
```

```
tid = 2
THREAD_BLOCKED
allelem
elem
...
```

```
tid = 3
THREAD_RUNNING
allelem
elem
...
```

```
tid = 4
THREAD_RUNNING
allelem
elem
...
```
Pintos Kernel (7)

- List management in Pintos
  - `#include <list.h> /* src/lib/kernel/list.h */`
  - A type oblivious, easy-to-use, circularly-linked list

```c
struct list {
    struct list *prev;  // Previous element
    struct list *next;  // Next element
}
```

```c
struct list_elem {
    struct list *prev;  // Previous element
    struct list *next;  // Next element
    // Additional elements...
}
```
**List management in Pintos (cont’d)**

- `list_init (struct list *list);`
  - Initializes list as an empty list
- `list_push_front (struct list *list, struct list_elem *elem);`
  - `list_push_back (struct list *list, struct list_elem *elem);`
  - Inserts elem at the beginning (end) of list
- `list_remove (struct list_elem *elem);`
  - Removes elem from its list
- `list_pop_front (struct list *list);`
  - `list_pop_back (struct list *list);`
  - Removes the front (back) element from list and returns it
- `list_entry (LIST_ELEM, STRUCT, MEMBER);`
  - Converts pointer to list element LIST_ELEM into a pointer to the structure that LIST_ELEM is embedded inside.
List management example

- Display thread list (tid & name)

```c
struct list all_list;

struct thread {
    tid_t tid;
    char name[16];
    ...
    struct list_elem allelem;
    ...
};
```

```c
void list_thread () {
    struct list_elem *e;

    for (e = list_begin(&all_list);
         e != list_end(&all_list);
         e = list_next(e))
    {
        struct thread *t =
            list_entry (e, struct thread, allelem);
        printf (“%d: %s
”, t->tid, t->name);
    }
}
```

- (cf.) [http://isis.poly.edu/kulesh/stuff/src/klist/](http://isis.poly.edu/kulesh/stuff/src/klist/)
Project 1: Threads

- **Requirements**
  - Alarm clock
  - Priority scheduling
  - Priority donation
  - Advanced scheduler

- **Test cases to pass (total 27 tests)**
- **Project 1-1 : alarm-single ~ priority-change**
Reimplement `timer_sleep()`

```c
void timer_sleep (int64 x);
```

- Suspends execution of the calling thread until time has advanced at least \( x \) timer ticks
- The current version simply “busy waits.”
  - The thread spins in a loop checking the current time and calling `thread_yield()` until enough time has gone by.
- Reimplement it to avoid busy waiting
- You don’t have to worry about the overflow of timer values.
**Time management in Pintos**

- On every timer interrupt, the global variable `ticks` is increased by one
  - The variable `ticks` represent the number of timer ticks since the Pintos booted
  - Timer frequency: `TIMER_FREQ` (= 100) ticks per second (defined in `<src/devices/timer.h>`)  
- The time slice is set to `TIME_SLICE` (= 4) ticks for each thread (defined in `<src/threads/thread.c>`)  
- `timer_interrupt()`: Timer interrupt handler
  - Increase the `ticks` variable
  - If the current thread has exhausted its time slice, call `thread_yield()`.
The current timer_sleep() implementation

- In <src/devices/timer.c>
- timer_ticks() returns the current value of ticks

```c
int64_t timer_elapsed (int64_t then)
{
    return timer_ticks () - then;
}

void timer_sleep (int64_t ticks)
{
    int64_t start = timer_ticks ();

    ASSERT (intr_get_level () == INTR_ON);

    while (timer_elapsed (start) < ticks)
        thread_yield ();
}
```
• Make a new list of threads ("waiting_list")
• Remove the calling thread from the ready list and insert it into the "waiting_list" changing its status to THREAD_BLOCKED
• The thread waits in the "waiting_list" until the timer expires
• When a timer interrupt occurs, move the thread back to the ready list if its timer has expired.
• Use <list.h> for list manipulation
Priority Scheduling (1)

- **Scheduling**
  - The scheduling policy decides which thread to run next, given a set of runnable threads

- **The current Pintos scheduling policy: Round-robin (RR) scheduling**
  - The ready queue is treated as a circular FIFO queue
  - Each thread is given a time slice (or time quantum)
    - `TIME_SLICE (= 4)` ticks by default
  - If the time slice expires, the current thread is moved to the end of the ready queue
  - The next thread in the ready queue is scheduled
  - No priority: All the threads are treated equally
/* Yields the CPU. The current thread is not put to sleep and may be scheduled again immediately at the scheduler's whim. */

void thread_yield (void)
{
    struct thread *cur = thread_current ();
    enum intr_level old_level;

    ASSERT (!intr_context ());

    old_level = intr_disable ();
    if (cur != idle_thread)
        list_push_back (&ready_list, &cur->elem);
    cur->status = THREAD_READY;
    schedule ();
    intr_set_level (old_level);
}
Priority Scheduling (3)

- The current Pintos scheduling (cont’d)

```c
/* Schedules a new process. At entry, interrupts must be off and
the running process's state must have been changed from
running to some other state. This function finds another
thread to run and switches to it.

It's not safe to call printf() until schedule_tail() has
completed. */

static void
schedule (void)
{
    struct thread *cur = running_thread ();
    struct thread *next = next_thread_to_run ();
    struct thread *prev = NULL;

    ASSERT (intr_get_level () == INTR_OFF);
    ASSERT (cur->status != THREAD_RUNNING);
    ASSERT (is_thread (next));

    if (cur != next)
        prev = switch_threads (cur, next);
    schedule_tail (prev);
}
```
Priority Scheduling (4)

- The current Pintos scheduling (cont’d)

/* Chooses and returns the next thread to be scheduled. Should return a thread from the run queue, unless the run queue is empty. (If the running thread can continue running, then it will be in the run queue.) If the run queue is empty, return idle_thread. */

static struct thread *
next_thread_to_run (void)
{
    if (list_empty (&ready_list))
        return idle_thread;
    else
        return list_entry (list_pop_front (&ready_list), struct thread, elem);
}
Priority Scheduling (5)

- **Priority scheduling**
  - Each thread is given a scheduling priority
  - The scheduler chooses the thread with the highest priority in the ready queue to run next
  - Thread priorities in Pintos
    - 64 priority levels (default = 31)
    - Max priority = 63
    - Min priority = 0
    - The initial priority is passed as an argument to `thread_create()`
    - The change to Highest priority should preempts other thread immediately
Applying jitter value

- **Overview**
  - Set timer interrupt in bochs at random intervals
    - Apply `-j seed` option at pintos running
      $ pintos -j 10 -- run alarm-multiple
  - In make operation
    - See line 56 in src/tests/Make.tests
      » TESTCMD += $(PINTOSOPTS)
    - When you run ‘make check’ or ‘make grade’, use this option
      $ make check PINTOSOPTS=''-j seed’
      $ make grade PINTOSOPTS=''-j seed’
  - Those are well described on 4~5p. PintOS documentation.
**Lock**

- `lock_acquire / lock_release`
  - Mutex lock (use semaphore)
  - threads/synch.h

- **Interrupt off**
  - Assume single core

```c
/* Yields the CPU. The current thread is not put to sleep and may be scheduled again immediately at the scheduler's whim. */
void
thread_yield (void)
{
  struct thread *cur = thread_current () ;
  enum intr_level old_level ;

  ASSERT (!intr_context ());

  old_level = intr_disable () ;
  if (cur != idle_thread )
    list_push_back (&ready_list , &cur->elem ) ;
  cur->status = THREAD_READY ;
  schedule () ;
  intr_set_level (old_level ) ;
}
```
Submission

- Due date: 9/28 23:59:59
- Check 7 tests (~priority_change) in 15 seconds
  - Include compile time
- Submission to sys.skku.edu
- [GroupNumber]_Project1-1.tar.gz
  - Ex) tar czf 1_project1-1.tar.gz src
- Only source code in this submission
- Do not copy